Table des matières

[1- Popular frameworks research (summer week 1) 2](#_Toc29298532)

[2- Mining scripts for issues and comments (summer week 1 and 2) 2](#_Toc29298533)

[3- Issues manual reading and Keywords search reading (summer weeks 2-6) 3](#_Toc29298534)

[4- Frameworks installation documentation (summer weeks 4 and 5) 4](#_Toc29298535)

[5- Call frequency of a bugfix (from summer week 7) 5](#_Toc29298536)

[5.1- Existing tracers (week 7) 5](#_Toc29298537)

[5.2- C++ tracer code (summer week 7-8) 5](#_Toc29298538)

[5.3- C++ syntax analyzers (summer week 7-10) 5](#_Toc29298539)

[5.4- Python inserter of the trace call (summer weeks 8-12) 6](#_Toc29298540)

[6- PyTorch issues reorganization (fall week 1) 6](#_Toc29298541)

[PyTorch building and training (fall week 2-8) 7](#_Toc29298542)

[PyTorch training with trace calls (fall week 8) 7](#_Toc29298543)

[Add more models to training (fall week 3) 8](#_Toc29298544)

[Parameters checker (fall week 3-5 for params checker) 8](#_Toc29298545)

[Mini tracer and python debug tool (fall week 6) 8](#_Toc29298546)

[(fall week 9) 8](#_Toc29298547)

[(fall week 10) 9](#_Toc29298548)

[(fall week 11) 9](#_Toc29298549)

[(fall week 12) 10](#_Toc29298550)

# 1- Popular frameworks research (summer week 1)

**Document 1:** **NN most popular libraries.docx** (<https://drive.google.com/open?id=1XH2S-7eN1wl2EYqiN9XQjaOBKwkfuyGq> )

The first step of the study consists of gathering neural network most popular frameworks. After some research on the internet, most results suggested these frameworks:

1) TensorFlow

2) Keras

3) PyTorch

4) Caffe

5) Theano

6) MXNET

7) CNTK

8) DeepLearning4J

9) Caffe2

10) Chainer

11) FastAI

And others

Reference : <https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a>

I referenced other websites in the google document. I also noted each framework’s Github, bug repository and website.

# 2- Mining scripts for issues and comments (summer week 1 and 2)

**Document 2:** **ml-framework-bugs\script\_issues.py**

From a csv containing frameworks’ information, mines each framework issues (with label or not) and saves them in a json and a csv file.

**Document 3: ml-framework-bugs\script\_comments.py**

From the issues’ csv, mines all comments of all issues in the csv and saves them in a json and a csv file. The labeled issues’ comments will be already in their own csv, since labeled issues are already separated after using script\_issues.py.

# 3- Issues manual reading and Keywords search reading (summer weeks 2-6)

*I apologize for the formatting of the tables. Be mindful of the table name, as some “relevant bugs” tables are long.*

**Google Drive folder :** [**https://drive.google.com/open?id=1nSMmAo0kiAlpdIVdbZFeolyICWfQNycH**](https://drive.google.com/open?id=1nSMmAo0kiAlpdIVdbZFeolyICWfQNycH)

**Document 4: Manually reading all frameworks** (<https://drive.google.com/open?id=15KPcTNVlmCPgZum-dQTl-JN7oG2lMZkQ_uss6LdeA7s> )

* TensorFlow (page 1):
* Relevant: manual reading of recent relevant issues. Brief look at the repo to find keywords/aspects some for the classification of bugs.
* Non-relevant: manual reading of recent non-relevant issues. Brief look at the repo to find some keywords/aspects for the classification of bugs.
* Caffe (page 2):
* Relevant:
  + Legitimate issues: manual reading of recent relevant issues. Brief look at the repo to find some keywords/aspects for the classification of bugs.
  + [label:bug] Exhaustive manual reading of Caffe issues with label:bug. The “Our Notes” columns contains Emilio’s opinion of the impact of bugs, which is a better indicator than the grade in the “Issue title” column. From #3254, “Our Notes” columns’ is an estimation of the bug impact. #297 and #284 are attempts to classify issues in the Deep learning stages.
* Non-relevant (page 8): One example of crashing bug. Three possible examples of PR probably having a minor impact on models.
* PR: is not a list of PRs, but a list of keywords.
* Sonnet (page 10):
* Relevant: exhaustive reading of all closed issues. Emilio’s notes in “Our Notes” column.
* Non-relevant: one non-relevant issue that was confusing. No Emilio notes.
* PR: No Emilio notes. I started looking at Sonnet’s PRs, but I did not continue.
* Swift for TensorFlow (page 12):
* Relevant: exhaustive reading of all closed issues. No Emilio notes.
* Non-relevant: non-relevant issues that were confusing. No Emilio notes.
* PR: empty.

**Document 5: manually reading Keras** (<https://drive.google.com/open?id=1ZJHPlkg1C0d9IOj3f6SpoegnSaG1TbGzQBfDgl19umQ> )

* Keras relevant issues (page 1):
* Relevant: manual reading of recent closed relevant issues. Covers a fewer number of issues than PyTorch. I started to focus on Pytorch because of its clearer version’s documentation. No Emilio notes.

**Document 6: Manually reading PyTorch (S2019 version)** (<https://drive.google.com/open?id=1m-pJxy1R00Gm4Vvi2lHc6bksKjqL8fmdVxOiaKfG-qo> )

* This document contains commit numbers for almost all issues/commits noted. The version number is also noted for those that was easier to retrace.
* PyTorch relevant issues (page 1):
* Relevant: manual reading of recent closed relevant issues. Covers a good number of issues, but probably not all, because I also used keywords. Most of the issues come from code>releases tabs on GitHub. No Emilio notes.
* PyTorch keywords issues (page 4):
* Relevant: manual reading of numerous issues found using grep and gitlog with keyword “bug” (and possibly “fix” and “bugfix” too…). I read a certain number of issues, then skipped a number of issues to “randomize” the reading. No Emilio notes.
* PyTorch files history, suggestion from the 9th of July meeting (page 10):
* Relevant: exhaustive reading of changes history for conv.py, batchnorm.py, maxpooling.py, pixelshuffle.py and pooling.py. No interesting results for maxpooling and pooling. No Emilio notes, but he said #12952’s previous commit causes a crash.

# 4- Frameworks installation documentation (summer weeks 4 and 5)

**Google drive folder :** [**https://drive.google.com/open?id=1lOnB1Av7gzzz6vXt\_XXx6iiXVIRLYiZq**](https://drive.google.com/open?id=1lOnB1Av7gzzz6vXt_XXx6iiXVIRLYiZq)

**Document 7:** **releases version support TensorFlow** (<https://docs.google.com/document/d/19T5njgSxdc74wnznbd3BssWKeVLZNSwq2WlBhU26PQc/edit>)

1) TensorFlow versions’ compatibility (gpu) (page 1): dependencies’ version for each TensorFlow gpu version. The table describes which dependencies are needed to install a buggy version.

2) whl packages for each Python version compatible with TensorFlow 1.13 (page 2): whl packages are easier for the installation of the buggy version. I think this table is for TF 1.14, not 1.13 …

3) whl packages for each Python version compatible with TensorFlow 1.14 (page 3): whl packages are easier for the installation of the buggy version.

4) whl packages for each Python version compatible with TensorFlow 1.12 (page 6): whl packages are easier for the installation of the buggy version.

5) whl packages for each Python version compatible with TensorFlow 1.11 (page 8): whl packages are easier for the installation of the buggy version.

6) whl packages for each Python version compatible with TensorFlow 1.10 (page 11): whl packages are easier for the installation of the buggy version.

7) whl packages for each Python version compatible with TensorFlow 1.9 (page 13): whl packages are easier for the installation of the buggy version.

8) whl packages for each Python version compatible with TensorFlow 1.8 (page 15): whl packages are easier for the installation of the buggy version.

The tables for version 1.7 and earlier are not present because the study focuses on bugs corrected from year 2016 and after.

**Document 8: releases version support PyTorch, Caffe and Theano (**<https://docs.google.com/document/d/13JBxRsZd4wkD4ep2BjeJ_Cz8v081srwaAgvoNFsGWvs/edit>**)**

1) PyTorch versions’ compatibility (page 1): dependencies’ version for each PyTorch version. The table describes which dependencies are needed to install a buggy version.

2) whl packages for each Python version compatible with various PyTorch versions (page 2): whl packages are more convenient for the installation of the buggy version. NOTE: The links are for CUDA 7.5

3) Caffe versions’ compatibility (page 4): dependencies’ version for each PyTorch version. The describes which dependencies are needed to install a buggy version.

4) whl packages for each Python version compatible with various Caffe versions (page 4): whl packages are more convenient for the installation of the buggy version.

5) Theano versions’ compatibility (page 5): dependencies’ version for each PyTorch version. The table describes which dependencies are needed to install a buggy version.

6) whl packages for each Python version compatible with various Theano versions (page 5): whl packages are more convenient for the installation of the buggy version.

**Document 9: meeting notes (**[**https://drive.google.com/open?id=1QKWxlSCWA1x8Q611SBamsuXsP\_\_ZtayrkmOCj6AE\_eE**](https://drive.google.com/open?id=1QKWxlSCWA1x8Q611SBamsuXsP__ZtayrkmOCj6AE_eE) **)**

Produced by Emilio. Summarizes all subjects for 17th June 2019, 24th June 2019, 1st July 2019 and 16th July 2019 meetings. A diagram at page 4 describes the workflow used for the research process of the study

# 5- Call frequency of a bugfix (from summer week 7)

## 5.1- Existing tracers (week 7)

Python seems to have existing tools for tracing. C language doesn’t seem too.

<https://docs.python.org/2/library/trace.html>

<https://docs.python.org/2/library/traceback.html>

<https://pymotw.com/2/trace/>

## 5.2- C++ tracer code (summer week 7-8)

Files are in **ml-framework-bugs\C Tracer.**

The integration of this code may cause compilation problems related to links edition. Emilio has done work regarding this.

## 5.3- C++ syntax analyzers (summer week 7-10)

CastXML: <https://github.com/CastXML/CastXML>

<https://github.com/thewtex/CastXMLSuperbuild>

GCC-XML: <https://github.com/gccxml/gccxml>

CastXML Is the maintained version of GCC-XML. Using the Superbuild is much simpler. If you wish to build from source, you will need to install Clang and LLVM … You might want to read these guides to build them:

<http://clang.llvm.org/get_started.html>

<https://www.llvm.org/docs/CMake.html>

## 5.4- Python inserter of trace call (summer weeks 8-12)

The code was reworked to improve its maintainability and clarity. The most recent version is now in the Python inserter of trace call version 2 (fall week 12) section.

# 6- PyTorch issues reorganization (fall week 1)

Document 12: manually reading PyTorch (<https://drive.google.com/open?id=1m-pJxy1R00Gm4Vvi2lHc6bksKjqL8fmdVxOiaKfG-qo>)

This document contains more issues notes and analysis for PyTorch releases 1.2, 1.1 and keywords search commits.

Table 12.1: Bugs to classify. Contains all bugs of the current *analysis*. Each commit is colored after the predicted effect of the bug

Red: No effect is expected to appear or had not appeared after running the issue.

Yellow: More analysis is needed, the appearance of an effect is unsure

Green: Great chances of having an effect from running the bug

Table 12.2: To analyze. Contains bugs that have great chances of having an effect. Green-labeled bugs from the table 12.1 should be copied into this table. These bugs will be executed in before the table 12.1’s bugs.

Table 12.3: Experiment logs. Contains bug’s version building information and runs information. Each entry from table 12.2 should be integrated into this table. Important instruction: choose an unique “Experiment Name” and mark down your name and build date for each bug. ***Look at run metrics for practical results of the runs.***

# PyTorch building and training (fall week 2-8)

PyTorch builder: git repository ml-framework-bugs/auto-launcher (<https://github.com/EmilioRivera/ml-framework-bugs/blob/master/auto-launcher/README.md>)

More specifically <https://github.com/EmilioRivera/ml-framework-bugs/blob/master/auto-launcher/README.md#launching-a-container-for-building> for the command to launch the PyTorch docker builder. Step by step instructions can be found in /home/kacham/Documents/pyt\_build\_steps.txt. The instructions tell in order which commands should be executed after launching the docker builder. The docker builder produces a whl (executable PyTorch package) for the commit checked out.

## PyTorch training with trace calls (fall week 8)

Trace calls were added to changed lines of code to demark the code parts that were called. To know if a bug is called, trace calls were added to the changed code. These trace calls write a output message in tracelogs when reached. At this point, I updated /home/kacham/Documents/pyt\_build\_steps.txt to follow the added steps for trace inserting. Basically, after launching the build docker, you need to checkout the commit of the bug, insert trace calls and build PyTorch. The resulting whl will be composed of the inserted PyTorch code.

Example:

Mini tracer: The trace call template and examples are in /home/kacham/Documents/ml-framework-bugs/C\_Tracer/mini\_tracer/main.py for Python and Documents\ml-framework-bugs\C\_Tracer\mini\_tracer\main.cpp for C++. Documents\ml-framework-bugs\C\_Tracer\mini\_tracer\results contains text outputs of the trace call in the main.py and main.cpp files. It can be deleted.

EXAMPLE:

PyTorch training: git repository ml-frameworks-evaluation (<https://github.com/EmilioRivera/ml-frameworks-evaluation/blob/master/README.md>) contains instruction for training of models with PyTorch. For training, step by step instructions can be found in /home/kacham/Documents/pyt\_build\_steps.txt.

The client can be launched by running /home/kacham/Documents/ml-frameworks-evaluation/training\_run.sh directly. training\_run.sh makes one training container for each .env file in the BASE\_DIR path. The env files can be found at /home/kacham/Documents/ml-frameworks-evaluation/envs. Each contains settings for training and for output files names. The most important settings for output files names are BUG\_NAME, CLIENT\_MANUAL\_DEPENDENCY and EVALUATION\_TYPE. CLIENT\_MANUAL\_DEPENDENCY is the commit SHA and it is used to launch the whl of the commit built.

The server is an executable docker image. Launching a docker container will automatically start the server. The training result will be written in the “results” docker volume.

A backup of training metrics is saved at /home/kacham/Documents/2019-10-24\_trainings\_backup. Those results were created from 2 runs training on EvaluationNet, EvaluationVGG and EvaluationAlex. There might be another backup in the “share” shared repository.

# Add more models to training (fall week 3)

Adapt python models in the code. From the source code at <https://pytorch.org/docs/stable/torchvision/models.html>, implement new models in /home/kacham/Documents/ml-frameworks-evaluation/src/client/models/pytorch\_models.py. EvaluationVGG and \_VGGG were implemented from PyTorch vision source code and EvaluationModel base class. It is recommended to follow the overall implementation of EvaluationVGG and \_VGGG for the other models.

# Parameters checker (fall week 3-5 for params checker)

Params\_checker: A little script to confirm the consistency of model initialization (/home/kacham/Documents/ml-frameworks-evaluation/src/client/params\_checker.py and /home/kacham/Documents/ml-frameworks-evaluation/src/client/params\_checker\_test.py for unit tests)

# Mini tracer and python debug tool (fall week 6)

Mini tracer: See “Mini tracer” in week 2-8 section.

Debug tool: Breakpoints were used to help finding the code parts that are called. Python source debugger was pdb (<https://docs.python.org/3/library/pdb.html>). C debugger was composed of sigint signals (<https://stackoverflow.com/a/4326474/9876427>), but this doesn’t work well, the program is interrupted and cannot retake after the interrupt.

# Sklearn defects analysis (fall week 9-12)

Sklearn framework work pipeline for training (<https://drive.google.com/open?id=1Sx_4l9dTnvF_YYJwbDe_KNH-J4aWDjpR_qJykWikzX8>) : The first few sections contain links for study and analysis. Releases links indicates important changes between versions and might reveal functional bugs. The source code links directs to some of the most used machine learning models. The repo bug label hadn’t been really showing functional bugs.

The tables coming after the links section are similarly structured as the manually reading PyTorch’s tables. Because of bugs cherrypicking, the “To analyze” table is less used; the bugs were directly tested. Choose a unique experiment name for each experiment in “Experiments log”.

Keyword search and history search: to find new bugs by cherry picking commits in the history of the github repo. Git commands facilitate the filtering of commits.

Show the complete history (of only one file, even if it was renamed) :

git log --follow -p -- path-to-file

Look for a word group that was involved in the history of code changes (of only one file) :

git log -S'bar' -- foo.rb

Look for a word group in the history of commit names (of all files):

git log --grep=word

Build sklearn: Cython is needed to recompile the source code. Make sure that Cython is in the environment path. With Cython installed, only two instructions should be necessary to recompile the source.

Training: /home/kacham/Documents/sklearn\_build\_steps.txt gives instructions for the trace insertion and training process. Here are some precisions about some steps:

0) name the branch like following: experiment\_name\_buggy or experiment\_name\_corrected

2) cp -a ../put\_in\_sklearn/. .

Call this command from within the /home/kacham/Documents/scikit-learn directory. /home/kacham/Documents/put\_in\_sklearn contains

1- a gitignore to facilitate saving analysis changes.

2- template\_traces.py the general template for python trace calls. Remember to change the path of tracelog for not overwriting other tracelogs.

3- train.py the training script. Script for discrete values is on the first half while the script for continuous values is on the second half. Remember to change the paths of tracelogs at each issue analyzed.

# Python inserter of trace call version 2 (fall week 12)

Project hierarchy:

The inserter.py file is the main file of the inserter project. For the commit given in parameter, the inserter adds trace calls at the changed lines. The inserter uses diff\_executor, then diff\_processor, file\_opener, analyzer\_syntax and analyzer\_content.

Diff\_executor file gets the filepaths of files to trace. Diff\_processor uses git diff to obtain the numbers of the changed lines. File\_opener gets the text of changed files in array format. Analyzer\_syntax checks syntax of the changed files. Analyzer\_indentation checks and add indentation for each trace call that is inserted.

Code utilization:

Command prompt call: python inserter.py (commit\_sha)

Example: python python\_inserter.py 9e5819a

Precisions about the code:

All lists share a certain dimension correspondancy. Lines\_number, files\_contents\_lines, to\_be\_inserted\_files\_lines and insertable\_lines first dimension’s length will be the number of changed files at the commit. Lines\_number and insertable\_lines second dimensions’ (each changed lines number) length will be the same and files\_contents\_lines and to\_be\_inserted\_files\_lines second dimensions’ (each file lines) length will be the same.

All lines\_numbers and indexes in code are the real line number -1. Example: if the real changed line number is 138, the changed line index in the code will be 137.

Each *[MAIN]* tag indicates the main function of each module.

Maintenance :

1) Code improvements and bug fixes:

Code improvements:

* Adapt the analyzer\_syntax code to C syntax
* Skip insertion of .pxi, .pyx, etc. and non-python files
* Insert trace calls on the line before or the line after
* Git diff returns the beginning of the changed code section, but we might prefer all the lines in the changed code section
* Integrate adaptive model, branch name, file name and line number in the trace calls
* Add one level of indentation after block starts or def
* Empty lines are generalized too “insertable”, but it might be too broad